A Simple Channel Vocoder

DSP Final Project – Coleman Jenkins (ctj9qgq)

Introduction

In the first semester of my third year, I took a class in the music department (MUSI 4545) with Luke Dahl. The focus of the class was designing audio effect plugins in C++. One algorithm he mentioned (but we didn't implement) was a channel vocoder, and I thought it would be fun to implement using techniques learned in DSP and MUSI 4545. A vocoder makes a synthesized instrument sound like a human voice.

It takes audio two signals as input and produces one audio signal as output. The first input (typically a voice) is analyzed for its spectral contents, and they are mapped on to the second audio signal as band pass filter amplitudes. This emulates a human voice because of the formants contained in vowels, which are resonant frequency peaks [1]. Mapping these resonant frequency peaks onto the second signal makes the second signal sound like a voice with the underlying frequency components of the original signal. A helpful diagram from Professor Dahl that I used for my vocoder implementation can be seen in Figure 1.

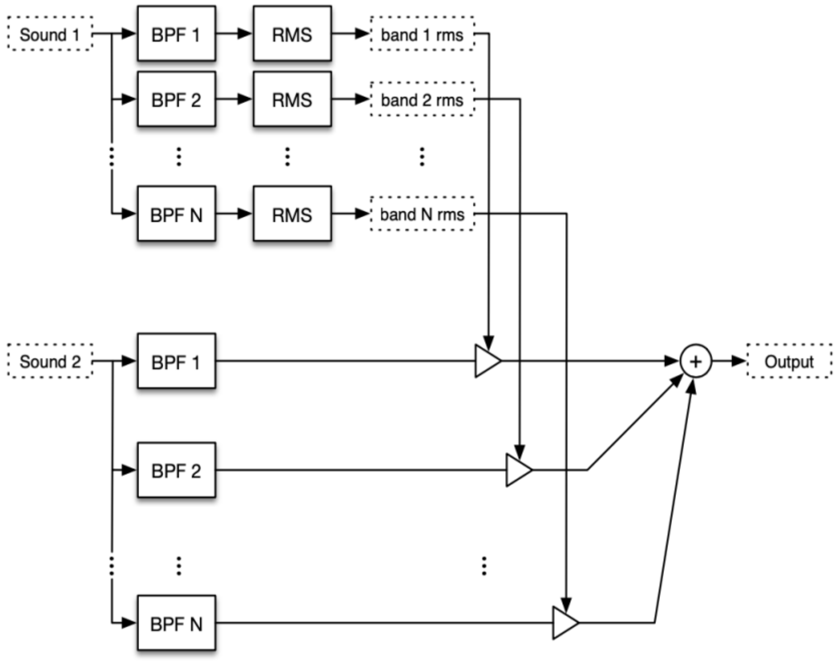

Figure 1: Vocoder Signal Diagram (From Prof. Dahl)

DSP Methods

The vocoder is based on two DSP techniques: band pass filters and envelope followers.

Band Pass Filters

My band pass filters were created by composing two biquad filters: a low pass and a high pass. The equation for a biquad is shown below:

$$H(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{a_0 + a_1 z^{-1} + a_2 z^{-2}}$$

In the time domain, this is equivalent to

$$y[n] = \frac{1}{a_0}(b_0 x[n] + b_1 x[n-1] + b_2 x[n-2] - a_1 y[n-1] - a_2 y[n-2])$$

The filters are multiplied in the z-domain:

$$H_{BP}(z) = H_{LP}(z) H_{HP}(z)$$

So the two systems are applied one after the other in the time domain.

The coefficients were determined using higher level parameters: f_c, the corner frequency; Q, the quality factor; and f_s, the sampling frequency based on ST AN2874 [2]. The following variables and coefficients apply for both the high pass and low pass filters:

$$\theta_C = \frac{2\pi f_c}{f_s}, K = \tan\left(\frac{\theta_C}{2}\right), W = K^2, \alpha = 1 + \frac{K}{Q} + W$$

$$a_0 = 1, a_1 = 2\frac{W-1}{\alpha}, a_2 = \frac{1 - \frac{K}{Q} + W}{\alpha}$$

The b coefficients for the high pass are:

$$b_0 = \frac{1}{\alpha}, b_1 = -\frac{2}{\alpha}, b_2 = b_0$$

The b coefficients for the low pass are:

$$b_0 = \frac{W}{\alpha}, b_1 = \frac{2W}{\alpha}, b_2 = b_0$$

The corner frequencies of the respective filters are determined by the number of bands requested. They are spaced out logarithmically because of their importance to pitch, which is perceived logarithmically. The following equations are used to determine the high pass corner frequency and the low pass corner frequency:

minimum frequency = 20 (Hz), maximum frequency = 15000 (Hz)

base = (maximum frequency / minimum frequency)^(1 / number of bands)

$$f_{c_{HP}} = minimum\ frequency * base^{band+0.1}$$

$$f_{c_{LP}} = minimum\ frequency * base^{band+0.9}$$

Band is the band index (0 to n_bands – 1) and the reason the bands are pushed inwards by 0.1 is such that they don't overlap too much.

By using a high Q for the biquads that make up the component filters, I was able to get a flatter pass band with more rejection of exterior frequencies, which can be seen in Figure 2 and compared to Figure 3.

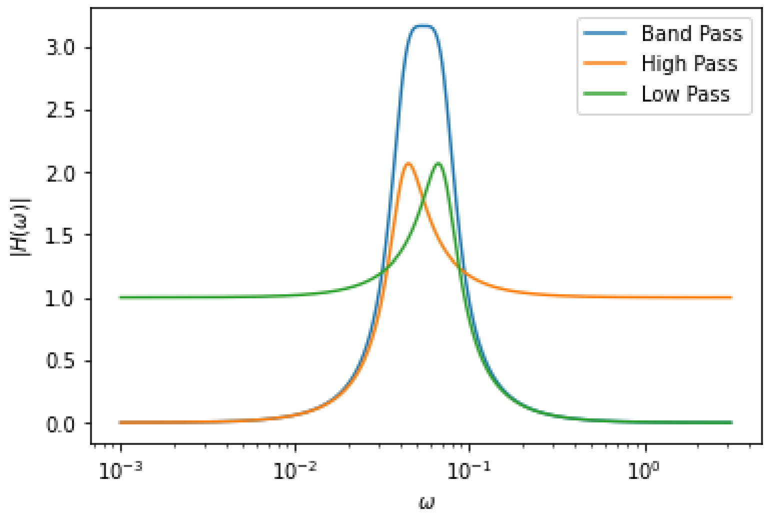

Figure 2: Band Pass with Component Filters Q=2

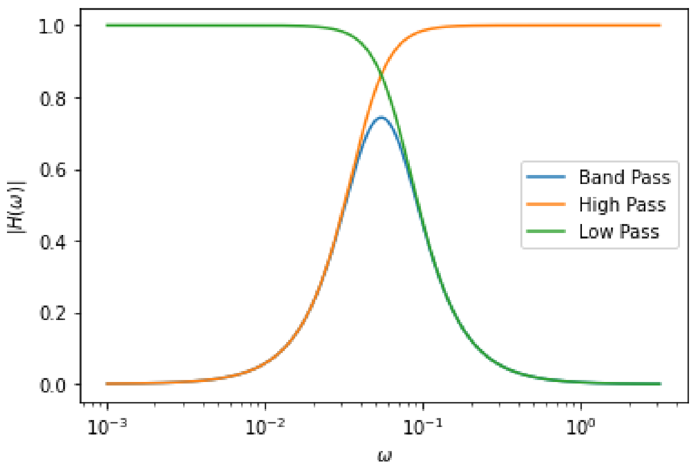

Figure 3: Band Pass with Component Filters Q=0.707

To test the band pass filter, one with $f_{c_{HP}} = 300\ Hz$ and $f_{c_{LP}} = 500\ Hz$ was applied to a female singer and the result clearly shows the narrow band.

All the band pass filters together with 10 bands can be seen in Figure 4.

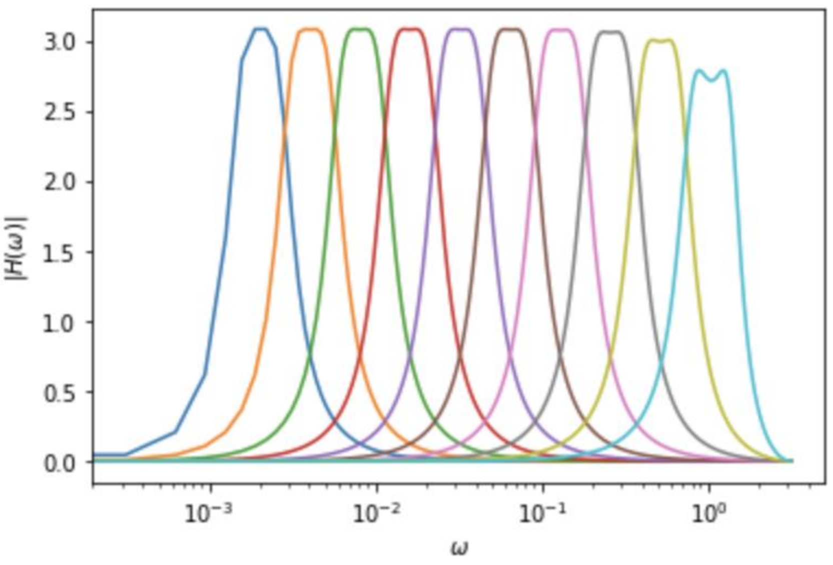

Figure 4: All Band Pass Filters

The same band pass filters are applied to signal 1 and signal 2 (though they are separate instances, the coefficients are the same). Also note that the magnitude of the band is greater than one. In a real time system it would need to be one, but since Python normalizes audio signals I left it this way.

Envelope Followers

In order to not have the band pass filter magnitudes of sound 1 map directly onto sound 2 (and thus making it sound choppy and unnatural), I used envelope followers for each band in signal 1. An envelope follower (also referred to as exponential smoothing [3]) tracks the amplitude of the signal slower than sample rate. It is a type of low pass filter. The equation for an envelope follower is

$$y[n] = bx[n] + (1 - b)y[n - 1], b = 1 - e^{\frac{1}{\tau f_s}}$$

Where τ is the decay time in seconds. Because this takes negative and positive values (and digital audio signals oscillate about 0), I used root mean square outside of the envelope follower. So, before $y[n]$, the actual input sample is squared to get $x[n]$, and after being put through the system, the resulting value returned is the square root of the output $y[n]$. The method can be seen applied to an audio signal in Figure 5. This is using a τ of 100 ms ($\tau = 0.1$).

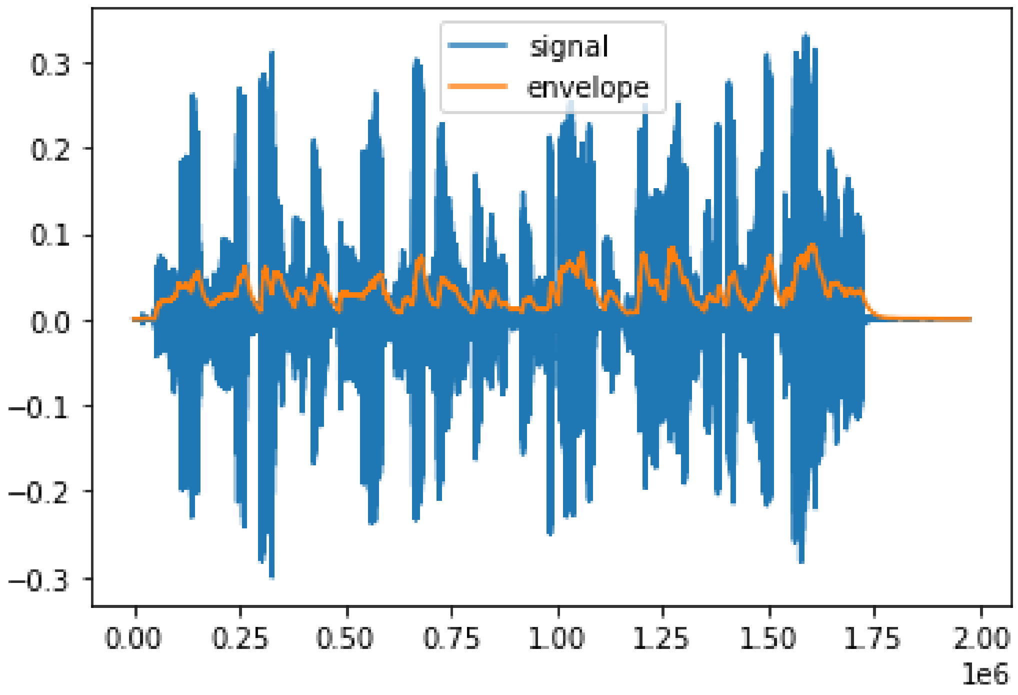

Figure 5: Envelope Follower Plot

Results

In order to test the vocoder, I used a voice recording of a female singer (from Professor Dahl) and a synth recording I made using Logic Pro [4]. The result certainly sounds like a vocoder and I think it turned out well. The singer can clearly be heard through the synthesizer signal, and it sounds musically interesting. From the intermediate results and the auditory verification of the final result, it is clear the algorithm is working as intended.

Challenges and Experimentation

At first, I tried to do the envelope follower without RMS and was really confused why it was staying around zero the whole time, but then I realized the reason was the negative inputs and why RMS was important.

In Prof. Dahl's class he mentioned a technique called "whitening", in which an envelope follower for each band of the music signal is used and the BPF outputs are divided by the envelope follower magnitude. This made the sound more even, but I didn't prefer it to the version without whitening so I left it out. An example of the result can be heard here.

The vocoder didn't sound too much like a voice at first, so I experimented with the band pass filters and the main things that helped were adding more bands (5 from 10) and making the component Qs larger to have a flatter band. I also tried linear spacing of the bands, but that didn't sound very good.

Finally, I experimented with multiple values of τ for the envelope followers. The sweet spot I found (10ms) responded fast enough to the vocal changes, but not too fast that it sounded choppy.

Sources

[1] https://www.britannica.com/science/phonetics/Vowel-formants

[2] https://www.st.com/resource/en/application_note/an2874-bqd-filter-design-equations-stmicroelectronics.pdf

[3] https://en.wikipedia.org/wiki/Exponential_smoothing

[4] https://www.apple.com/logic-pro/